


Craig Williams and Paul F. Reynolds Jr.
University of Virginia

Bronis R. de Supinski
Lawrence Livermore National Laboratory

 The authors describe a class of directory coherence protocols called delta coherence protocols. These protocols use network guarantees to support a new and highly concurrent approach to maintaining a consistent shared memory.

Delta Coherence Protocols

Caching data can reduce access latency and improve data availability, but it also raises the problem of how to maintain consistency among copies of writable data. The problem appears in different guises in different contexts: it appears as the cache coherence problem in multiprocessors, as the problem of maintaining

a distributed shared memory in distributed computations, and as the replica control problem in distributed databases. This article describes the *home update protocol*, a member of the class of coherence protocols called *delta coherence protocols* that uses *isotach guarantees*¹ to solve the coherence problem in a new and highly concurrent way. (Due to space constraints, and to avoid obscuring the basic concept of the protocols, we describe the protocol at a high level and do not address practical implementation issues.) Our goal is to show how isotach guarantees are useful in solving the coherence problem and in reasoning about coherence protocols.

The coherence problem is difficult, because it requires coordinating events across nodes. The traditional approach to the problem is to reduce the coordination required by limiting concurrency or weakening the correctness criteria. Hardware-based coherence protocols are traditionally divided into two classes:² *snoopy protocols*, which require a shared bus, and *directory protocols*, intended for point-to-point networks. A shared bus serializes memory requests. This serialization readily yields an agreed total order among requests, but it limits concurrency and

scalability. Directory protocols are more scalable, but existing directory protocols that enforce sequential consistency (SC) require that nodes execute requests one at a time and invalidate or lock copies while executing write requests.

Delta protocols use isotach guarantees to coordinate accesses, an approach that lets delta protocols enforce SC without limiting concurrency. However, delta protocols require isotach guarantees. Whether delta coherence protocols outperform existing protocols depends on the cost of implementing isotach guarantees and on the extent to which applications can take advantage of the high level of concurrency delta protocols offer.

Isotach systems

An isotach (Greek translation: *iso*, same; *tach*, speed) system implements a logical time system¹ in which all messages appear to travel at the same speed—one unit of logical distance per unit of logical time. Given this property, called the *isotach invariant*, a processor can control the logical receive time of a message it sends by controlling the logical send time.

Neighboring nodes in an isotach system

for copy c		
$\delta(c)$	The delta of c . In home update protocol, $dist(home, c)$	
for operation op executed on copy c		
$t_s(op)$	Send time of op	} related by isotach invariant: $t_r(op) = t_s(op) + d(op)$
$t_r(op)$	Receive time of op	
$d(op)$	Logical distance of op	
$t_x(op)$	Execution time of op	$t_x(op) = t_r(op)$, by assumption
$t_{effx}(op)$	Effective execution time of op	$t_{effx}(op) = t_r(op) - \delta(c)$
$x_{dist}(op)$	Execution distance of op	$x_{dist}(op) = t_{effx}(op) - t_s(op) = d(op) - \delta(c)$

Figure 1. Delta coherence protocol terms and notation.

exchange signals called *tokens* to implement a distributed logical clock. The *pulse* at a processor is the number of tokens the processor has received. An isotach logical time is a lexicographically ordered three-tuple in which the first and most significant component is the pulse at the processor where the event occurs. The remaining two components, the *process identifier* (pid) and *rank*, are tie-breakers used to order send and receive events that occur in the same pulse. The sender pid orders events with identical pulse components. The rank—or issue order—orders events with identical pulse and pid components.

The isotach logical time system extends Leslie Lamport's logical time system³ by guaranteeing that send and receive times are consistent with the isotach invariant: each message travels one unit of logical distance per pulse of logical time. Isotach systems can implement a variety of distance metrics.⁴ Here, $dist(p, p')$ —the logical distance from node p to node p' —is the routing distance from p to p' —that is, the number of switches traversed by a message that p sends to p' . For any message m that p sends to p' , $d(m)$ —the logical distance message m travels—is $dist(p, p')$. For simplicity, we assume distances are static. Distances can be asymmetric—that is, $dist(p, p')$ does not necessarily equal $dist(p', p)$. By the isotach invariant, for any message m , m 's logical receive time is exactly $d(m)$ pulses after m 's logical send time, so $t_r(m) = t_s(m) + d(m)$. (The scalar quantity $d(m)$ is added to the tuple $t_s(m)$ by adding $d(m)$ to the tuple's pulse component.) We assume processors execute messages in receive order. Thus, for any message m , $t_x(m)$ — m 's logical execution time—equals $t_r(m)$.

This assumption is for simplicity and is stronger than necessary.⁵

Most delta protocols require an isotach system that supports *predictable responses*. A predictable response is a message m' sent in response to another message m such that we can predict the send time of m' from the receive time of m : $t_s(m') = t_r(m) + c$. For simplicity, we assume c is 0. (In any practical system, c is a small tunable system constant, greater than zero.) Given the isotach invariant and knowledge of the logical distances involved, we can predict the receive time of m' from the send time of m : $t_r(m') = t_s(m) + d(m) + d(m')$. A predictable response inherits the original message's pid and rank components.

Each processor has a switch interface unit (SIU) that tracks logical time and acts as the interface between applications and the isotach system. An application can assume that the isotach system will appear to execute its messages in the order issued. Given the isotach invariant and the assumption that the system will execute messages in the order received, an SIU can control the relative order in which locally issued messages appear to be executed. In particular, an SIU can ensure that a batch of locally issued messages appear to be executed at the same time by sending the messages so that the destinations receive the messages in the same logical pulse. An SIU can also ensure that messages issued in a sequence appear to be executed in that sequence by sending the messages so that the destinations receive the messages in nondecreasing pulses.

Isotach systems can be implemented using the *isonet* algorithm, in which network switches route messages in logical time order.¹ Alternatively, an implementation can shift the work of ordering mes-

sages to the SIUs to permit the use of commodity switches. The Isotach Project at the University of Virginia has implemented a prototype system based on this approach on a cluster of commodity PCs connected with Myrinet.⁶ Both algorithms are scalable, requiring the exchange of tokens only among nearest neighbors. In the prototype, which implements isotach functionality in software, the roundtrip,

user-to-user-level latency of isotach messages is on the order of 50 μ sec, about twice that of nonisotach messages on the same hardware.⁷ To further reduce the cost of maintaining isotach guarantees, we are redesigning the messaging-layer software and building a second-generation prototype with custom SIUs.

Model

The coherence problem occurs in several contexts, each with its own terminology. The terms used here are from the literature on cache coherence. We rely on the reader interested in DSM or replica control to make the appropriate translations.

We consider a system consisting of multiple processors connected to a memory system. The memory system encapsulates the representation of shared memory and the procedures for accessing it. The processor–memory system interface is as follows:

- Processors issue read and write requests to the memory system. A read request (**READ**) on v instructs the memory system to return the value of v ; a write request (**WRITE**) on variable v instructs the memory system to assign a specified value to v . A variable is shared if more than one processor can issue requests on it. We consider only shared variables.
- The memory system returns a value in response to each **READ**.

Internal details of the memory system are not visible to the processors.

A memory system consists of interconnected memories and controllers pro-

grammed to execute a coherence protocol. The memory space is partitioned across the memory modules (MMs). Each processor has a cache memory and cache controller (CC), which manages the cache and translates locally issued requests into *operations*. An operation reads, writes, creates, or destroys a copy of a variable. The CC generates one or more write operations (writes) for each WRITE and a single read operation (read) for each READ. The phrase “the execution of request R on copy c ” means “the execution of the operation resulting from R that is executed on copy c .” In a delta protocol, the CC also acts as the SIU—that is, it tracks logical time and controls the logical send times of locally issued operations.

For each variable v , the primary copy—called the *home copy*—is located in an MM. The MM containing v 's home copy is v 's *home*. Secondary copies—called *cache copies*—are located in the cache memories. In a *static copyset* protocol, the number and locations of cache copies are determined statically. In a *dynamic copyset* protocol, the memory system can create and destroy cache copies. A request for v is a *hit* if a copy of v is in the issuing processor's cache; otherwise it is a *miss*.

In a delta protocol, the memory system sends each operation as an isotach message. The logical distance and the send, receive, and execution times of an operation are those of the message carrying the operation. An operation on the local cache copy is sent as a *self-message*—an isotach message that a processor sends to itself. Because self-messages do not enter the network, for any self-message m , $d(m) = 0$ and $t_r(m) = t_s(m)$. Figure 1 summarizes terms relevant to operations in a delta protocol.

Each copy c in a delta protocol has a delta, denoted $\delta(c)$. In the home update protocol, $\delta(c) = \text{dist}(\text{home}, c)$ —the number of logical time pulses required to propagate an update and thus the number of pulses by which c lags behind the home copy. The delta of a home copy is zero. For any operation op on c , $t_{\text{effx}}(\text{op})$, op 's effective execution time is $t_x(\text{op}) - \delta(c)$. Informally, $t_{\text{effx}}(\text{op})$ is op 's apparent execution time—its logical execution time adjusted to compensate for its delta. The

Related work

Coherence protocols are notoriously hard to prove correct. Logical time has provided a formal yet intuitive basis for reasoning about coherence¹ and has been widely used in reasoning about protocols that enforce relaxed consistency semantics.

Message-ordering guarantees have been used elsewhere to increase coherence-protocol concurrency. Race-free networks² enforce Sequential Consistency with fewer restrictions on concurrency by restricting the network topology and the paths taken by updates. Multicast-snooping protocols³ use isotach guarantees to eliminate acknowledgment messages. Protocols can use totally ordered multicasts to ensure that WRITES execute atomically and can use causally ordered messages to ensure execution is consistent with program order.⁴

References

1. A.E. Condon et al., “Using Lamport Clocks to Reason about Relaxed Memory Models,” *Proc Fifth Int'l Symp. High Performance Computer Architecture*, IEEE Press, Piscataway, N.J., 1999.
2. A. Landin, E. Hagersten, and S. Haridi, “Race-Free Interconnection Networks and Multiprocessor Consistency,” *Proc. 18th Int'l Symp. Computer Architecture*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1991, pp. 106–115.
3. E.E. Bilir et al., “Multicast Snooping: A New Coherence Method Using a Multicast Address Network,” *Proc. 26th Int'l Symp. Computer Architecture*, in *Computer Architecture News*, Vol. 27, No. 2, 1999, pp. 294–304.
4. A. Fekete, M.F. Kaashoek, and N. Lynch, “Implementing Sequentially Consistent Shared Objects Using Broadcast and Point-to-Point Communication,” *J. ACM*, Vol. 45, No. 1, Jan. 1998, pp. 35–69.

execution distance of op , $x_{\text{dist}}(\text{op})$, is defined as $t_{\text{effx}}(\text{op}) - t_s(\text{op})$. Thus, $x_{\text{dist}}(\text{op}) = d(\text{op}) - \delta(c)$, where c is the copy on which the memory system executes op .

Correctness criteria

A coherence protocol's most basic task is to make replication transparent to the processors (see the “Related work” sidebar). The result of any execution should be as if the requests of the processors were executed on a single-copy memory—that is, on a memory containing a single copy of each variable. Coherence protocols can enforce the following ordering properties:

- *Sequential consistency*. A memory system enforces SC if “[t]he result of any execution is as if the [requests] of all the processors were executed in some sequential order, and the [requests] of each individual processor appear in this sequence in the order specified by the program.”⁸ The execution shown in Figure 2a violates SC, because no sequential ordering of the requests can produce the results shown. In Figure 2b, even though P1's accesses are executed out of order, the execution shown is SC, because it produces the same

results as a sequential execution in which P1's accesses are executed in program order followed by P2's accesses in program order.

- *Atomicity*. Requests issued as part of the same transaction or atomic action are executed so that they appear to be executed as an indivisible unit. Thus, the result of any execution is as if the requests of all the processors were executed in some sequential order and the requests in each transaction appear as a contiguous subsequence, not interleaved with other requests.

We use the term *atomicity* to mean consistency (not failure) atomicity; the guarantee is about the relative order in which requests appear to be executed, not about the results of a failure. A protocol that enforces *failure atomicity* ensures that all requests in the same transaction are executed on an all-or-nothing basis. A fault-free system is normally assumed in the context of multiprocessor cache coherence and is often left to separate mechanisms in the DSM context. The isotach prototype uses a sender-based protocol and a reliable network (Myrinet) to achieve reliable communication. An unreliable network would require using a commit protocol.

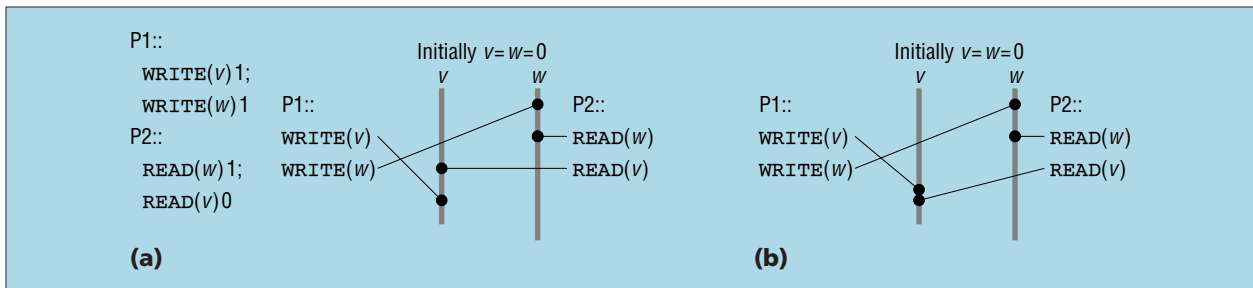


Figure 2. (a) Violating *sequential consistency*; (b) even though P1's accesses are executed out of order, the execution is SC.

With a few exceptions, cache coherence protocols for multiprocessors and DSM protocols focus on SC (or a weaker variant) and leave the task of enforcing atomicity to separate mechanisms. On the other hand, databases focus on enforcing atomicity. The high cost of enforcing SC and atomicity has led to extensive exploration of weaker memory consistency models. Whether the resulting improvement in performance justifies the more complex memory interface is an undecided issue.⁹ Delta protocols enforce atomicity and SC using isotach ordering guarantees without the locks and restrictions on pipelining required in conventional systems. Thus, delta protocols represent an alternative to weakening the guarantees the memory system offers.

Home update delta coherence protocol

The home update protocol is the simplest of the delta protocols and serves as the basis for the other delta protocols, which include invalidate as well as update protocols. As its name indicates, it is an update protocol in which the home is responsible for distributing updates. The protocol is a directory protocol: for any variable v , the home for v stores a directory that records the set of processors with cache copies of v . For simplicity, we assume a bit-vector representation for the directory: bit i in the bit vector for v is set if and only if processor i has a cache copy of v . (A system could use any of several tagged directory schemes for improving the scalability of directories instead.) We describe both a static copyset version of the protocol (static protocol) and a dynamic copyset version (dynamic protocol).

EXECUTING REQUESTS

A CC translates each locally issued request into one or more operations, called *initiating operations*. In the home update

protocol, each request results in exactly one initiating operation. Other actions the memory system takes in executing a request depend on the request type:

- **READ miss** on variable v . The CC generates and schedules a read on v 's home copy (we describe scheduling later). On receiving the read, v 's home returns a read response. In the static protocol, a read response is simply a message, not an operation—there is no copy at the receiving CC on which the read response operates. On receiving the read response, the CC returns the value to p .
- **READ hit**. The CC generates and schedules a read on v 's cache copy. (Recall that an operation on the local copy is a self-message and has a logical send and receive time.) At the logical receive time, the CC executes the read on its cache copy, returning the value to p .
- **WRITE (hit or miss)**. The CC generates and schedules a write on v 's home copy. On receiving the write, v 's home assigns the value to the home copy and sends a write to every processor in v 's directory (including p if p is in v 's directory). Writes that the home sends are usually called *updates*. An *own-update* is an update a CC received in response to its own write. On receiving an update, a CC assigns the value to the cache copy.

IMPLEMENTING A DYNAMIC COPYSET

We can adapt the protocol so it can create and destroy cache copies:⁵

- A CC destroys its copy of v by sending a release message to v 's home. The home executes the release by removing p from v 's directory.
- A CC creates a cache copy as a result of a miss. When the home receives an operation on v that p sends, it adds p to v 's directory. When a CC with no

valid cache copy of v executes a read response or an own-update on v , it creates a cache copy.

USING ISOTACH GUARANTEES

The home update protocol, as described so far, is similar to other update protocols. The protocol differs from others in its use of isotach guarantees.

The isotach invariant lets each CC control the effective execution time of requests by scheduling the send times of initiating operations. The *scheduling* algorithm uses this control to enforce SC:

```
lastR = 0
for each request R issued by p
  if R is a READ hit xdist =
    dist(home, p)
  else xdist = dist(p, home);
  sendp = max(lastR - xdist, now);
  lastR = sendp;
```

The CC tracks `lastR`, the effective execution pulse of the last request it scheduled, and schedules the initiating operation of each new request so that its effective execution pulse is no less than `lastR`. The effective execution pulse is the pulse component of the effective execution time. A request with the same effective execution pulse as the previous request has a later effective execution time due to its rank component.

The home MM sends updates as predictable responses, allowing the definition of copy deltas and establishing the relationship between each variable's cache and home copies. As we show later, sending updates as predictable responses also ensures that all writes resulting from the same `WRITE` have the same effective execution time, with the result that each `WRITE` appears to execute atomically. Although the execution times of the writes can differ, their effective execution times are identical because

Static protocol

We show that for any execution E of any program P on a memory system M under the static protocol, there is an equivalent sequential execution S of P on a single-copy memory system M' , where S is sequential consistency.

Definition. For any request R , the effective execution time of R is the effective execution time of the initiating operation resulting from R .

Lemma 1. The effective execution times of requests derived from execution E of program P define a total order over the requests in P .

Proof. Because each logical time is a three-tuple in which the second and third components serve as tie-breakers, each initiating operation for a request in E has a unique effective execution time.

Definition. For any program P , let P' be the permutation of P in which the requests in P appear in increasing order by their effective execution times.

Definition. Let S be the execution of P in which the requests in P are executed on M' in the order in which the requests appear in P' .

Lemma 2. All operations resulting from the same request have the same effective execution time.

Proof. Because a READ results in only one operation, the claim is trivially true for READS. A WRITE results in an initiating write executed on the home copy and an update on each cache copy. The effective execution time of the initiating write w is $t_r(w)$, because the write is executed on the home copy and the delta of the home copy is zero. For any update u , the home sends to copy c' in response to w , $t_{\text{effx}}(u) = t_r(u) - \delta(c')$. Because u is a predictable response to operation op , $t_r(u) = t_r(w) + \text{dist}(\text{home}, c')$. Thus, $t_{\text{effx}}(u) = t_r(w) + \text{dist}(\text{home}, c') - \delta(c')$. Because $\delta(c') = \text{dist}(\text{home}, c')$, $t_{\text{effx}}(u) = t_{\text{effx}}(w)$.

Lemma 3. For any copy c in E or S and any two operations op and op' executed on c , op and op' are executed in the order of their effective execution times.

Proof. For all operations that M executes on any copy c in E , the difference between the effective execution time and the execution time is the same, $\delta(c)$. Thus, $t_{\text{effx}}(op)$ is less than $t_{\text{effx}}(op')$ if and only if $t_x(op)$ is less than $t_x(op')$ —in other words, M executes operations on c in increasing order by their effective execution times. By definition of S , M' executes all operations in S , including any operations executed on any given copy c , in increasing order of their effective execution times.

Lemma 4. For any two requests R and R' , if R is executed on copy c before R' , then R is executed before R' on every copy on which both R and R' are executed.

Proof. By Lemmas 2 and 3.

Lemma 5. Every READ in P returns the same value in S and E .

Proof. Consider any READ R' on any variable v in P . M executes R' on exactly one copy c . Let W be the immediately preceding WRITE on c —the WRITE that assigns the value M returns in response to R' . Because every request M executes in E is executed by M' in S , M' executes W and R' on the copy of v in S . By Lemma 4, because M executes W before R' on c in E , M' executes W before R' on the copy of v in S . We show by contradiction that there is no intervening WRITE W' between W and R' on the copy in S . Because, in the static protocol, M executes every WRITE on v on every copy of v , M executes W' on c . By Lemma 4, M executes W' on c between W and R' , contradicting the assumption that W is the immediately preceding write for R' in E . Thus, the same write is the immediately preceding write for R' in E and S and E and S return the same value for R' .

Lemma 6. Execution S is SC.

Proof. Consider any two requests R and R' issued by the same processor p , where R is issued before R' . Let op be the initiating operation for R and op' be the initiating operation for R' . By the scheduling algorithm, the CC chooses $t_s(op')$ such that $t_{\text{effx}}(op') > t_{\text{effx}}(op)$. Thus, R' appears after R in P' and is executed after R in S .

Theorem. The static protocol is correct.

Proof. By Lemma 5, E and S are equivalent. By Lemma 6, S is SC. Thus, the result of any execution on a memory system using the static protocol is the same as if it were executed on M' in some sequential order consistent with the program order.

Proof of the dynamic protocol requires showing that every read operation is executed on a copy that has received the immediately preceding write.¹

References

1. C. Williams, P.F. Reynold Jr., and B.R. de Supinski, *Delta Coherence Protocols: The Home Update Protocol*, Tech. Report CS-2000-08, Dept. of Computer Science, Univ. of Virginia, Charlottesville, Va., Mar. 2000.

each copy's delta exactly offsets the time required to propagate the update.

Correctness

We prove the correctness of the static protocol in the "Static protocol" sidebar. A proof of the full protocol appears elsewhere.⁵ We show that the result of any execution under the protocol is the same as it would be if it were executed on memory system M' , which is known to be correct. Memory system M' executes requests serially in some sequential order

on a single-copy memory, translating each request into a single operation. We show that for any execution E of any program P on a memory system that uses the static protocol, there is an equivalent sequential execution S of P on M' , where S is SC. A program is a sequence of requests, in the order in which they are submitted, and an execution of program P is the sequence of operations resulting from P in the order in which they are executed. Executions S and E of P are equivalent if every READ in S returns the same value as the corresponding READ in E .

Atomicity

We can adapt the home update protocol to execute batches of requests atomically. Exploiting this capability changes the programming model—instead of using locks or barriers to enforce atomicity, a processor issues batches of requests called *isochrons* and the memory system executes the requests in each isochron so that they appear to be executed at the same time. Because a processor must issue all the requests in an isochron as a batch, isochrons cannot contain internal data dependences.

However, we can implement atomic actions with internal data dependences, using isochrons together with a class of operations called *split operations*.¹⁰

Adapting the protocol to enforce atomicity requires changing only the scheduling algorithm. Each CC schedules requests so that all requests in the same isochron have the same effective execution pulse, and schedules each isochron so that it has an effective execution pulse no less than the previously scheduled isochron. We show requests in each isochron are executed atomically by showing that the requests occur in the equivalent serial execution S as a contiguous subsequence.⁵ Because all requests in the same isochron have the same effective execution pulse and are issued by the same processor as a batch, no other request can have an intervening effective execution time.

IN DELTA PROTOCOLS, each copy c has a delta, $\delta(c)$, equal to the number of logical pulses by which the copy lags behind the home copy. The deltas let nodes control the order in which requests appear to execute and facilitate proving delta protocols correct.

Delta coherence protocols use isotach guarantees to enforce SC with fewer restrictions on concurrency than existing protocols. First of all, under these protocols, the memory system can pipeline requests. Existing protocols that enforce SC require that the execution of a request not start until the execution of the previous request issued by the same processor completes.¹¹ (Sarita V. Adve and Mark D. Hill have proposed an SC protocol that lets nodes overlap the execution of a **WRITE** with another request, with a restriction that the effect of the second request cannot be visible to any node until after the **WRITE** is globally performed.¹²) Delta protocols can overlap the execution of requests, requiring only that a request not appear to complete before the previous request completes—in other

words, that its effective execution time not precede that of the previous request.

Second, delta coherence protocols don't require acknowledgments. Existing protocols use acknowledgments to inform a node when its **WRITE** completes. Relying on acknowledgments adds message traffic and, more importantly, increases latency—a node delays executing a request not just until the completion of the previous request, but until it receives acknowledgment of the completion. In delta protocols, a node determines from local information the completion time of each request before it sends the initiating operation.

Third, multiple processors can write the same variable concurrently. Invalidate protocols do not permit concurrent writes, though update protocols do, subject to the restriction that writes are not immediately readable.

Fourth, writes are immediately readable. In the absence of strong message-ordering guarantees, existing protocols that ensure SC cannot return the value of a read to a cache copy until the **WRITE** that supplied that value is globally performed—that is, until all cache copies are updated or invalidated.¹¹ This requirement is easy to satisfy in invalidation protocols but difficult in update protocols.

Finally, processors can execute multiple requests atomically without locks. Most existing protocols that enforce atomicity use two-phase locking. Alternatively, protocols can assign transactions timestamps and abort and restart any transactions that cannot be executed in timestamp order. Delta protocols let a processor access multiple variables atomically without locks or restarts. Processors can execute isochrons without synchronizing or obtaining exclusive access to the variables accessed.

Delta protocols offer a significantly higher level of concurrency than existing coherence protocols, while a prototype isotach network implementation demonstrates that the cost of providing this additional concurrency is low. We expect delta protocols to be useful in applications that maintain many copies of data items or in which data contention is high. //

ACKNOWLEDGMENTS

NSF grant CCR-9503143 supported this work, with additional funding provided under DARPA grant DABT63-95-C-0081. Portions of the work were performed under the auspices of the US Department of Energy by the University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48; UCRL-JC-137415.

References

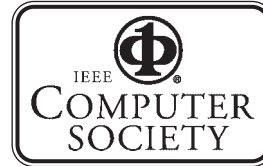
1. P.F. Reynolds, Jr., C. Williams, and R.R. Wagner Jr., "Isotach Networks," *IEEE Trans. Parallel and Distributed Systems*, Vol. 8, No. 4, Apr. 1997, pp. 337–348.
2. D.J. Lilja, "Cache Coherence in Large-Scale Shared-Memory Multiprocessors: Issues and Comparison," *ACM Computing Surveys*, Vol. 25, No. 3, Sept. 1993, pp. 303–338.
3. L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM*, Vol. 21, No. 7, July 1978, pp. 558–565.
4. B.R. de Supinski, *Logical Time Coherence Maintenance*, doctoral dissertation, Dept. Computer Science, Univ. of Virginia, Charlottesville, Va., 1998.
5. C. Williams, P.F. Reynolds Jr., and B.R. de Supinski, *Delta Coherence Protocols: The Home Update Protocol*, Tech. Report CS-2000-08, Dept. Computer Science, Univ. of Virginia, Charlottesville, Va., Mar. 2000.
6. N.J. Boden et al., "Myrinet: A Gigabit-Per-Second Local-Area Network," *IEEE Micro*, Vol. 15, No. 1, Feb. 1995, pp. 29–36.
7. J. Regehr, *An Isotach Implementation for Myrinet*, Tech. Report CS-97-12, Dept. Computer Science, Univ. of Virginia, May 1997.
8. L. Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocessor Programs," *IEEE Trans. Computers*, Vol. C-28, No. 9, Sept. 1979, pp. 690–691.
9. M.D. Hill, "Multiprocessors Should Support Simple Memory-Consistency Models," *Computer*, Vol. 31, No. 8, Aug. 1998, pp. 28–34.
10. C. Williams, *Concurrency Control in Asynchronous Computations*, doctoral dissertation, Dept. Computer Science, Univ. of Virginia, 1993.
11. S.V. Adve and K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial," *Computer*, Vol. 29, No. 12, Dec. 1996, pp. 66–76.
12. S.V. Adve and M. Hill, "Weak Ordering—A New Definition," *Proc. 17th Int'l Symp. Computer Architecture*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1990, pp. 15–26.

Craig Williams is a senior scientist in the Computer Science Department of the University of Virginia. Her research interests include parallel data structures and both the hardware and software aspects of parallel computation. She received her BA in economics from the College of William and Mary, a JD from Columbia Law School, and an MS in computer science. She received her PhD in computer science at the University of Virginia. Contact her at the Univ. of Virginia, Computer Science Dept., 151 Engineer's Way, Charlottesville, VA 22904; ccw4s@virginia.edu.

Paul F. Reynolds Jr. is a professor of computer science at the University of Virginia. He received his PhD from the University of Texas. He has published widely in the area of parallel and distributed computation, specifically in parallel and distributed simulation and in parallel language and algorithm design. He has been a consultant to numerous corporations and government agencies in the systems and simulation areas. Contact him at the Univ. of Virginia, Computer Science Dept., 151 Engineer's Way, Charlottesville, VA 22904; reynolds@virginia.edu.

Bronis R. de Supinski is a computer scientist in the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory. His research interests include cache coherence and distributed shared memory, consistency semantics, networks of workstations, distributed object computing, and parallel and distributed simulation. He earned his PhD in computer science from the University of Virginia. His projects include a performance analysis of several programming models on SMP clusters and the design and evaluation of MPI collective communications that use multiple low-level communication mechanisms. He is a member of the ACM and IEEE Computer Society. Contact him at Lawrence Livermore National Laboratory, Center for Applied Scientific Computing, 7000 East Ave., Livermore, CA 94550; bronis@llnl.gov.

PURPOSE The IEEE Computer Society is the world's largest association of computing professionals, and is the leading provider of technical information in the field.



MEMBERSHIP Members receive the monthly magazine **COMPUTER**, discounts, and opportunities to serve (all activities are led by volunteer members). Membership is open to all IEEE members, affiliate society members, and others interested in the computer field.

EXECUTIVE COMMITTEE

President: GUYLAINE M. POLLOCK*
Sandia National Laboratories
1515 Eubank SE
Bldg. 836, Room 2276
Organization 0049
Albuquerque, NM 87123

President-Elect: BENJAMIN W. WAH*
Past President: LEONARD L. TRIPP*
VP, Educational Activities: JAMES H. CROSS II*
VP, Conferences and Tutorials: WILLIS K. KING (1ST VP)*
VP, Chapters Activities: WILLIAM W. EVERETT*
VP, Publications: SALLIE V. SHEPPARD*
VP, Standards Activities: STEVEN L. DIAMOND (2ND VP)*

VP, Technical Activities: MICHEL ISRAEL*
Secretary: DEBORAH K. SCHERRER*
Treasurer: THOMAS W. WILLIAMS*
2000-2001 IEEE Division V Director:
DORIS L. CARVER*
1999-2000 IEEE Division VIII Director:
BARRY W. JOHNSON*
2001-2002 IEEE Division VIII Director:
BRUCE D. SHRIVER*
Acting Executive Director
ANNE MARIE KELLY

*voting member of the Board of Governors *nonvoting member of the Board of Governors

BOARD OF GOVERNORS

Term Expiring 2000: Fiorenza C. Albert-Howard, Paul L. Borrell, Carl K. Chang, Deborah M. Cooper, James H. Cross, II, Ming T. Liu, Christina M. Schober

Term Expiring 2001: Kenneth R. Anderson, Wolfgang K. Giloi, Haruhisa Ichikawa, Lowell G. Johnson, David G. McKendry, Anneliese von Mayrbauser, Thomas W. Williams

Term Expiring 2002: James D. Isaak, Gene F. Hoffnagle, Karl Reed, Deborah K. Scherrer, Kathleen M. Swigger, Ronald Waxman, Akibiko Yamada

Next Board Meeting: 10 November 2000, Dallas, Texas

COMPUTER SOCIETY OFFICES

Headquarters Office
1750 Massachusetts Ave. NW,
Washington, DC 20036-1992
Phone: +1 202 371 0101
Fax: +1 202 728 9614
E-mail: hq.ofc@computer.org

Publications Office
10662 Los Vaqueros Cir.,
PO Box 3014
Los Alamitos, CA 90720-1314
General Information:
Phone: +1 714 821 8380
membership@computer.org
Membership and
Publication Orders: +1 800 272 6657
Fax: +1 714 821 4641
E-mail: cs.books@computer.org

European Office
13, Ave. de L'Aquilon
B-1200 Brussels, Belgium
Phone: +32 2 770 21 98
Fax: +32 2 770 85 05
E-mail: euro.ofc@computer.org

Asia/Pacific Office
Watanabe Building
1-4-2 Minami-Aoyama,
Minato-ku, Tokyo 107-0062,
Japan
Phone: +81 3 3408 3118
Fax: +81 3 3408 3553
E-mail: tokyo.ofc@computer.org

EXECUTIVE STAFF

Acting Executive Director: ANNE MARIE KELLY
Chief Financial Officer: VIOLET S. DOAN

Publisher: ANGELA BURGESS
Manager, Research & Planning: JOHN C. KEATON

Acting Director, Volunteer Services: MARY-KATE RADA

IEEE OFFICERS

President: BRUCE A. EISENSTEIN
President-Elect: JOEL B. SNYDER
Executive Director: DANIEL J. SENESE
Secretary: DAVID J. KEMP
Treasurer: DAVID A. CONNOR
VP, Educational Activities: LYLE D. FEISEL
VP, Publications Activities: MICHAEL S. ADLER
VP, Regional Activities: ANTONIO BASTOS
VP, Standards Association: DONALD C. LOUGHRY
VP, Technical Activities: ROBERT A. DENT
President, IEEE-USA: MERRILL W. BUCKLEY JR.

